Multiple Clock Domains (MCD)

Arvind with Nirav Dave
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Plan

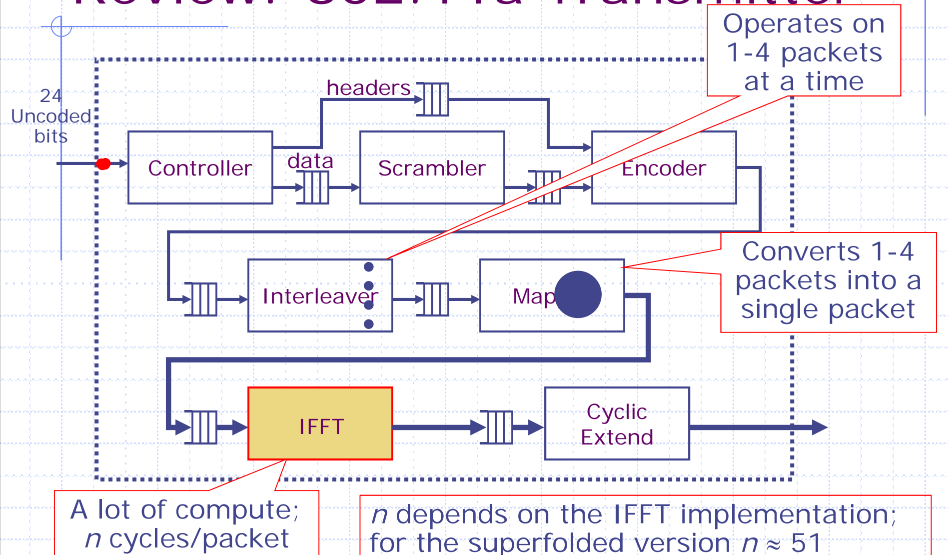
- Why Multiple Clock Domains
  - 802.11a as an example
- How to represent multiple clocks in Bluespec
- MCD Syntax
- Revisit 802.11a
- Synchronizers

Why Multiple Clock Domains

- Arise naturally in interfacing with the outside world
- Needed to manage clock skew
- Allow parts of the design to be isolated to do selective power gating and clock gating
- Reduce power and energy consumption

March 15, 2010 http://csg.csail.mit.edu/6.375 L12-3

Review: 802.11a Transmitter



March 15, 2010 http://csg.csail.mit.edu/6.375 L12-4

Synthesis results for different microachitectures

Design	Area (mm²)	Best CLK Period	Throughput CLK/symbol	Latency
Comb.	1.03	15 ns	1	15 ns
Pipelined	1.46	7 ns	1	21 ns
Folded	0.83	8 ns	3	24 ns
S Folded 1 Radix	0.23	8 ns	48-51	408 ns

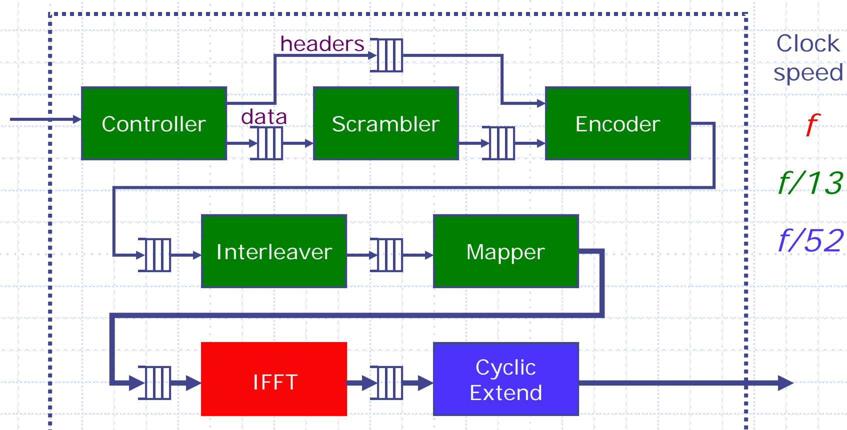
For the same throughput SF has to run ~16 times faster than F

TSMC .13 micron; numbers reported are before place and route.

Single radix-4 node design is ¼ the size of combinational design but still meets the throughput requirement easily; clock can be reduced to 15 - 20 Mhz

Dave,Pellauer,Ng 2005

Rate Matching



After the design you may discover the clocks of many boxes can be lowered without affecting the overall performance

Power-Area tradeoff

- The power equation: $P = \frac{1}{2} CV^2f$
  - V and f are not independent; one can lower the f by lowering V – linear in some limited range
- Typically we run the whole circuit at one voltage but can run different parts at different frequencies
- We can often increase the area, i.e., exploit more parallelism, and lower the frequency (power) for the same

One would actually want to explore many relative frequency partitionings to determine the real area/power tradeoff

Plan

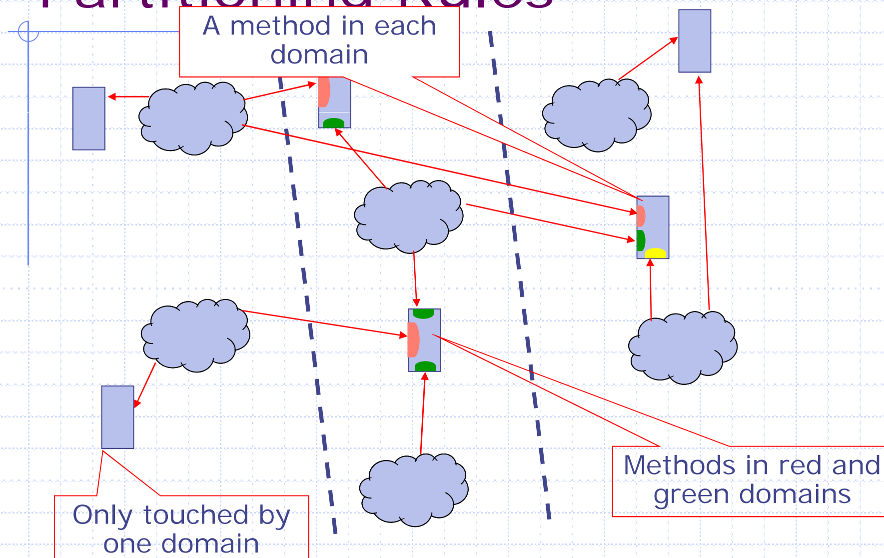
- Why Multiple Clock Domains
  - 802.11a as an example
- How to represent multiple clocks in Bluespec
- MCD Syntax
- Revisit 802.11a
- Synchronizers

Associating circuit parts with a particular clock

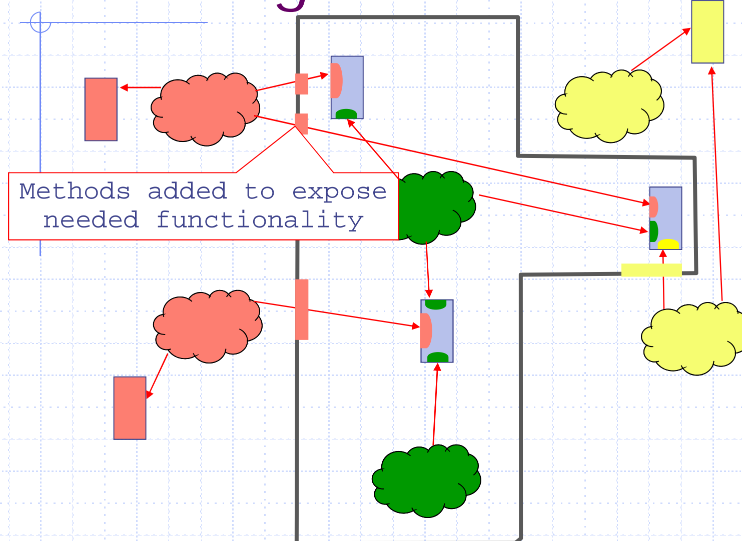
Two choices to split the design:

- Partition State
  - Rules must operate in multiple domains
- Partition Rules
  - State Elements must have methods in different clock domains

It is very difficult to maintain rule atomicity with multi-clock rules. Therefore we would not examine "Partitioned State" approach further



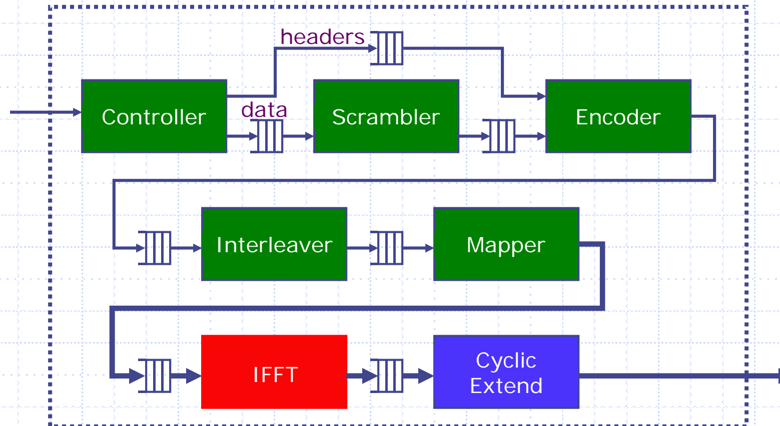
Handling Module Hierarchy



We need a primitive MCD synchronizer: for example

- enq one on clock and deq/first/pop on another
- full/empty signals are conservative approximations
  - may not be full when full signal is true
- We'll discuss implementations later

Back to the Transmitters



Domains in the Transmitter

```
let controller    <- mkController();
let scrambler     <- mkScrambler_48();
let conv_encoder  <- mkConvEncoder_24_48();
let interleaver   <- mkInterleaver();
let mapper        <- mkMapper_48_64();
let ifft          <- mkIFFT_Pipe();
let cyc_extender  <- mkCyclicExtender();
rule controller2scrambler(True);
    stitch(controller.getData,scrambler.fromControl);
endrule
```

These colors are just to remind us about domains

... *many such stitch rules* ...

```
function Action stitch(ActionValue#(a) x,
                       function Action f(a v));
   action  let v <- x;  f(v);   endaction
endfunction
```

Coloring the rules?

```
rule controller2scrambler(True);
 stitch(controller.getData,
        scrambler.fromControl);
endrule

rule scrambler2convEnc(True);
 stitch(scrambler.getData,
          conv_encoder.putData);
endrule

rule mapper2ifft(True);
 stitch(mapper.toIFFT, ifft.fromMapper);
endrule
```

All methods in the same domain

Using different domains…

Domain Crossing

```
rule mapper2ifft(True);
     stitch(mapper.toIFFT, ifft.fromMapper);
endrule
```

inline stitch

```
 rule mapper2ifft(True);
      let x <- mapper.toIFFT();
      ifft.fromMapper(x)
endrule
```

Different methods in an action are on different clocks – we need to change the clock domains

Introduce a domain crossing module

```
let m2ifftFF <- mkSyncFIFO(size,clkGreen, clkRed);
```

- Many such synchronizers
- In real syntax, one clock value is passed implicitly

Fixing the Domain Crossing

```
rule mapper2ifft(True);
  let x <- mapper.toIFFT();
  ifft.fromMapper(x)
endrule
```

```
rule mapper2fifo(True);
  stitch(mapper.toIFFT, m2ifftFF.enq);
endrule

rule fifo2ifft(True);
  stitch(pop(m2ifftFF), ifft.fromMapper);
endrule
```

```
let m2ifftFF <- mkSyncFIFO(size,clkGreen,clkRed);
```

synchronizer syntax is not quite correct

Similarly for IFFT to CyclicExt

```
let ifft2ceFF <- mkSyncFIFO(size,clkRed,clkBlue);

rule ifft2ff(True);
  stitch(ifft.toCyclicExtender, ifft2ceFF.enq);
endrule

rule ff2cyclicExtender(True);
  stitch(pop(ifft2ceFF), cyc_extender.fromIFFT);
endrule
```

Now each rule is associated with exactly one clock domain!

Plan

- Why Multiple Clock Domains
  - 802.11a as an example
- How to represent multiple clocks in Bluespec
- MCD Syntax
- Revisit 802.11a
- Synchronizers

How to introduce clocks

```
module mkTransmitter(Transmitter#(24,81));

  let controller   <- mkController();
  let scrambler    <- mkScrambler_48();
  let conv_encoder <- mkConvEncoder_24_48();
  let interleaver  <- mkInterleaver();
  let mapper       <- mkMapper_48_64();
  let ifft         <- mkIFFT_Pipe();
  let cyc_extender <- mkCyclicExtender();

  // rules to stitch these modules together
```

How should we
1. Generate different clocks?
2. Pass them to modules?
3. Introduce clock synchronizers and fix the rules?

Instantiating modules with clocks (clock is a type)

- Synthesized modules have an input port called CLK, which is passed to all interior instantiated modules by default
- However, any module can be instantiating with an explicit clock

```
Clock c = … ;
Reg# (Bool) b <- mkReg (True, clocked_by c);
```

- Modules can also take clocks as ordinary arguments, to be fed to interior module instantiations

The clockOf() function

makes the implicit clock explicit

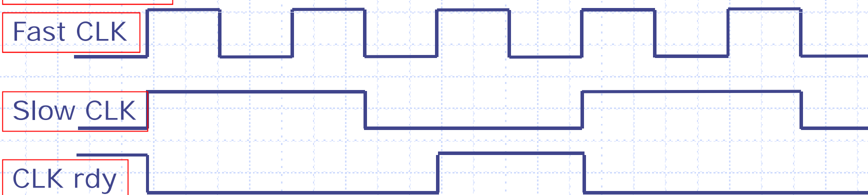
```
Reg# (UInt# (17)) x <- mkReg (0, clocked_by c);
Clock c0 <- exposeCurrentClock;
let y = x + 2;
Clock c1 = clockOf (x);
Clock c2 = clockOf (y);
```

- c, c0, c1 and c2 are all equal
- Can be used interchangeably for all purposes
- If the expression is a constant, the result is the special value noClock
  - noClock values can be used on in any domain

Clock Dividers

```
interface ClockDividerIfc ;
    interface Clock   fastClock ;  // original clock
    interface Clock   slowClock ;  // derived clock
    method    Bool    clockReady ;
endinterface

module mkClockDivider #( Integer divisor )
                        ( ClockDividerIfc ifc ) ;
```



Plan

- Why Multiple Clock Domains
  - 802.11a as an example
- How to represent multiple clocks in Bluespec
- MCD Syntax
- Revisit 802.11a
- Synchronizers

Step 1: Introduce Clocks

```
module mkTransmitter(Transmitter#(24,81));

  let clockdiv13 <- mkClockDivider(13);
  let clockdiv52 <- mkClockDivider(52);
  let clk13      = clockdiv13.slowClock;
  let clk52      = clockdiv52.slowClock;

  let controller    <- mkController();
  let scrambler     <- mkScrambler_48();
  let conv_encoder <- mkConvEncoder_24_48();
  let interleaver   <- mkInterleaver();
  let mapper        <- mkMapper_48_64();
  let ifft          <- mkIFFT_Pipe();
  let cyc_extender <- mkCyclicExtender();

  // rules to stitch these modules together
```

Step 2: Pass the Clocks

```
module mkTransmitter(Transmitter#(24,81));
   let clockdiv13 <- mkClockDivider(13);
   let clockdiv52 <- mkClockDivider(52);
   let clk13      = clockdiv13.slowClock;
   let clk52      = clockdiv52.slowClock;

   let controller   <- mkController(clocked_by clk13);
   let scrambler    <- mkScrambler_48(clocked_by clk13);
   let conv_encoder <- mkConvEncoder_24_48(clocked_by clk13);
   let interleaver  <- mkInterleaver(clocked_by clk13);
   let mapper       <- mkMapper_48_64(clocked_by clk13);
   let ifft         <- mkIFFT_Pipe();  <- Default Clock
   let cyc_extender <- mkCyclicExtender(clocked_by clk52);
   // rules to stitch these modules together
```

Now some of the stitch rules have become illegal because they call methods from different clock families

Introduce Clock Synchronizers

Step 3: Introduce Clock Synchronizers

```
module mkTransmitter(Transmitter#(24,81));
   let m2ifftFF   <- mkSyncFIFOToFast(2,clockdiv13);
   let ifft2ceSF  <- mkSyncFIFOToSlow(2,clockdiv52);
   …
   let mapper       <- mkMapper_48_64(clocked_by clk13);
   let ifft         <- mkIFFT_Pipe();
   let cyc_extender <- mkCyclicExtender(clocked_by clk52);
   rule mapper2fifo(True); //split mapper2ifft rule
        stitch(mapper.toIFFT, m2ifftFF.enq);
   endrule
   rule fifo2ifft(True);
        stitch(pop(m2ifftFF), ifft.fromMapper);
   endrule
   rule ifft2fifo(True); //split ifft2ce rule
        stitch(ifft.toCycExtend, ifft2ceFF.enq);
   endrule
   rule fifo2ce(True);
        stitch(pop(ifft2ceFF), cyc_extender.fromIFFT);
   endrule
```

Did not work...

stoy@forte: ~/examples/80211$ bsc -u -verilog Transmitter.bsv

Error: "./Interfaces.bi", line 62, column 15: (G0045)
Method getFromMAC is unusable because it is connected to a clock not available at the module boundary.

Need to fix the Transmitter's interface so that the outside world knows about the clocks that the interface methods operate on.

(These clocks were defined inside the module)

The Fix – pass the clocks out

```
interface Transmitter#(type inN, type out);
   method Action getFromMAC(TXMAC2ControllerInfo x);
   method Action getDataFromMAC(Data#(inN) x);

   method ActionValue#(MsgComplexFVec#(out))
                            toAnalogTX();

   interface Clock  clkMAC;
   interface Clock  clkAnalog;
endinterface
```

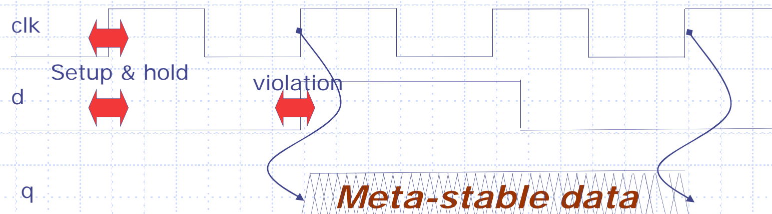
Clock Summary

- The *Clock* type, and type checking ensures that all circuits are clocked by actual clocks
- BSV provides ways to create, derive and manipulate clocks, safely
- BSV clocks are *gated*, and gating fits into Rule-enabling semantics (clock guards)
- BSV provides a full set of speed-independent data synchronizers, already tested and verified
  - The user can define new synchronizers
- BSV precludes unsynchronized domain crossings

Clock Synchronizers

Moving Data Across Clock Domains

- Data moved across clock domains appears asynchronous to the receiving (destination) domain
- Asynchronous data will cause meta-stability
- The only safe way: use a *synchronizer*

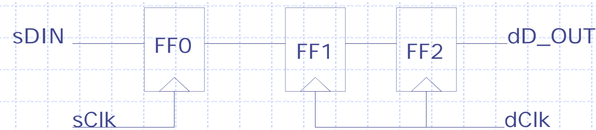


Synchronizers

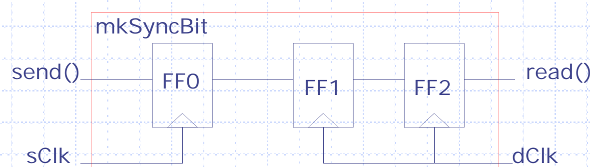
- Good synchronizer design and use reduces the probability of observing meta-stable data
- Bluespec delivers conservative (speed independent) synchronizers
- User can define and use new synchronizers
- Bluespec does not allow unsynchronized crossings (compiler static checking error)

2 - Flop BIT-Synchronizer

- Most common type of (bit) synchronizer
- FF1 will go meta-stable, but FF2 does not look at data until a clock period later, giving FF1 time to stabilize
- Limitations:
  - When moving from fast to slow clocks data may be overrun
  - Cannot synchronize words since bits may not be seen at same time



Bluespec's 2-Flop Bit-Synchronizer




```
interface SyncBitIfc ;
   method Action  send ( Bit#(1) bitData ) ;
   method Bit#(1) read () ;
endinterface
```

- The designer must follow the synchronizer design guidelines:
  - No logic between FF0 and FF1
  - No access to FF1's output

Use example: MCD Counter

- Up/down counter: Increments when up_down_bit is one; the up_down_bit is set from a different clock domain.
- Registers:

```
Reg# (Bit#(1)) up_down_bit <-
            mkReg(0, clocked_by ( writeClk ) );
Reg# (Bit# (32)) cntr <- mkReg(0); // Default Clk
```

- The Rule (attempt 1):

```
rule countup ( up_down_bit == 1 ) ;
      cntr <= cntr + 1;
endrule
```

Illegal Clock Domain Crossing

Adding the Synchronizer

```
SyncBitIfc  sync <- mkSyncBit( writeClk,
                     writeRst, currentClk ) ;
```

Split the rule into two rules where each rule operates in one clock domain

clocked by *writeClk*

```
rule transfer ( True ) ;
   sync.send ( up_down_bit );
endrule
```

clocked by *currentClk*

```
rule countup ( sync.read == 1 ) ;
   cntr <= cntr + 1;
endrule
```

MCD Counter

```
module mkTopLevel#(Clock writeClk, Reset writeRst)
                  (Top ifc);
Reg# (Bit# (1))  up_down_bit <- mkReg(0,
                                clocked_by(writeClk),
                                reset_by(writeRst)) ;
Reg# (Bit# (32)) cntr <- mkReg (0) ;
                                  // Default Clocking
Clock currentClk <- exposeCurrentClock ;
SyncBitIfc  sync <- mkSyncBit ( writeClk, writeRst,
                                currentClk ) ;
rule transfer ( True ) ;
   sync.send( up_down_bit );
endrule
rule countup ( sync.read == 1 ) ;
   cntr <= cntr + 1;
endrule
```

We won't worry about resets for the rest of this lecture

Different Synchronizers

- Bit Synchronizer
- FIFO Synchronizer
- Pulse Synchronizer
- Word Synchronizer
- Asynchronous RAM
- Null Synchronizer
- Reset Synchronizers

Documented in Reference Guide